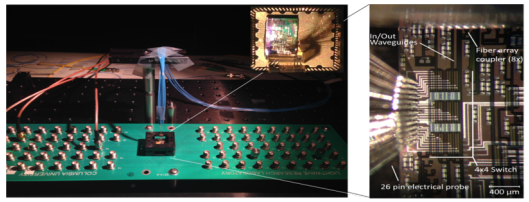
Modern problems require modern solutions: How modern CMake supports modern C++ in Kokkos



CMake++

PRESENTED BY

Jeremiah Wilke, Sandia National Labs, Livermore, CA

Contributors: Daniel Arndt, Damien Lebrun-Grandie, Jonathan Madsen, Nathan Ellingwood, Christian Trott

P3HPC, Virtual Kansas City, 2020 SAND2020-8713 C

Performance portability in the build system same goal as C++: single platform-agnostic "source" gives high-performance build

Platform-independent C++ through template metaprogramming

```
View<Scalar*> output("Output", N);
parallel_for(N, KOKKOS_LAMBDA(const int& i){
  // work on output
})
```

1. Generate functionally correct code, dispatch to appropriate device
2. Choose execution order/layout
3. Partition and map work to threads

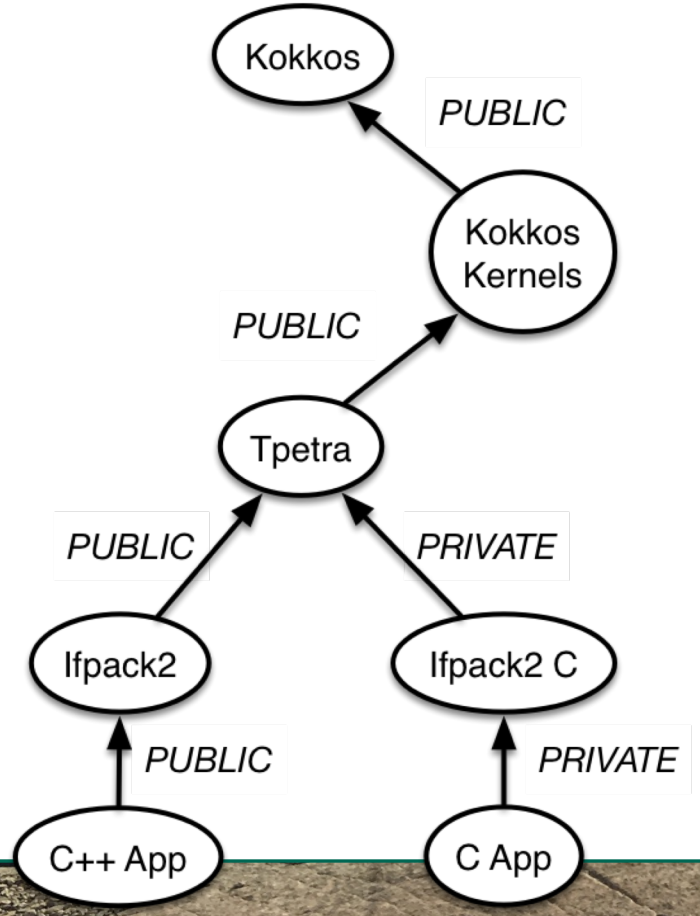
Platform-independent builds through CMake

```
find_package(Kokkos REQUIRED)
target_link_libraries(myTarget Kokkos::kokkos)
```

1. Generate functionally correct code, dispatch to appropriate device
2. Add architecture optimization flags
3. Tune compiler optimizations

Modern CMake/C++ wants a clean separation of 'building' and 'using' libraries: transitive targets and properties

- ***Building:*** What flags (includes, definitions, compile, link) does my project need to build?
- ***Using:*** What flags do downstream projects need when building with my project?



- Automake requires collecting and forwarding, e.g.
  `My_CXX_FLAGS += $(Kokkos_CXX_FLAGS)`
  - You can always do anything in Make if you try hard enough…
- `TARGET_LINK_LIBRARIES(Ifpack2)` makes C++ App depend transitively on Kokkos flags (PUBLIC)
- `TARGET_LINK_LIBRARIES(Ifpack2_C)` does not make C App depend transitively on Kokkos flags (PRIVATE)
- Transitive *static* and *dynamic* libraries handled

Modern CMake/C++ wants a clean separation of 'building' and 'using' libraries: transitive targets and properties

- ***Building:*** What flags (includes, definitions, compile, link) does my project need to build?
- ***Using:*** What flags do downstream projects need when building with my project?
- CMake 3, first "modern" version released June 2014
  - Clean separation of building and using
  - Targets and properties preferred over exporting variables
- Targets (executables or libraries) are very flexible
  - Link + header, header-only, link-only
  - Compile-flags only in case of OpenMP or pthreads, e.g.
  - Imported from external install or built within your project
- Flags (i.e. target properties) are declared as `PUBLIC` (build + use), `PRIVATE` (build), `INTERFACE` (use)
  - Property `INCLUDE_DIRECTORIES` specifies include flags for library being built
  - Property `INTERFACE_INCLUDE_DIRECTORIES` specifies flags for downstream projects

A "library" can be way more interesting that just a .h and .a/.so

Kokkos needs to propagate build flags to ALL downstream projects: Xeon/Volta70, GCC, OpenMP/CUDA example

`-march=skylake-avx512`
`-mtune=skylake-avx512`
`-mrtm`
	Skylake optimization flags

`-expt-extended-lambda`	Lambda support for Kokkos
`-arch=sm_70`	Build for Volta70
`-Xcompiler -fopenmp`	Build OpenMP support on host
`-std=c++11`	Resolve conflicting standard requests from different libs

Single case not terrible, but unrealistic to expect app developers to handle all use cases

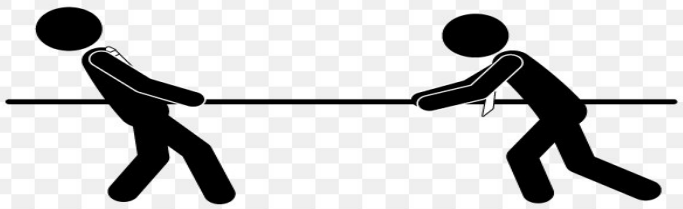
Corporate America wants its developers to never waste time worrying about build systems and build times

- Facebook: Buck (open-sourced)
  - Focus is incremental builds, write in Starlark
- Google: Blaze (open-sourced as Bazel)
  - Focused on monorepos, not great with transitive dependencies, write in Starlark
- Twitter has Pants, Amazon has Brazil
  - Can’t say much about these
- Open source Maven:
  - Java focused, write XML files, does handle transitive dependencies
- SCons, Waf:
  - Python-based, not great with transitive dependencies
- Meson:
  - Basically CMake in Python, good with transitive dependencies

Scalable development with CMake targets: Have your monorepo and eat it, too

AMESOS

kokkos



Trilinos

Smaller components

- Ease of reuse across community
- Better scaling of version control
- Improve collaboration with smaller dependencies

Monorepo

- Ease of reuse within a project
- Atomic commits
- Improve collaboration through flexible code ownership

Kokkos Goal: Be a central repository of tuning knowledge and share with the world! Need software engineering problem to maximize focus on performance issues and sharing components.

Scalable development with CMake targets: Have your monorepo and eat it, too

```
target_link_libraries(myTarget Kokkos::kokkos)
```

Downstream apps can be happily oblivious to where the target comes from

Could be Autools/Make installed project converted to import CMake target

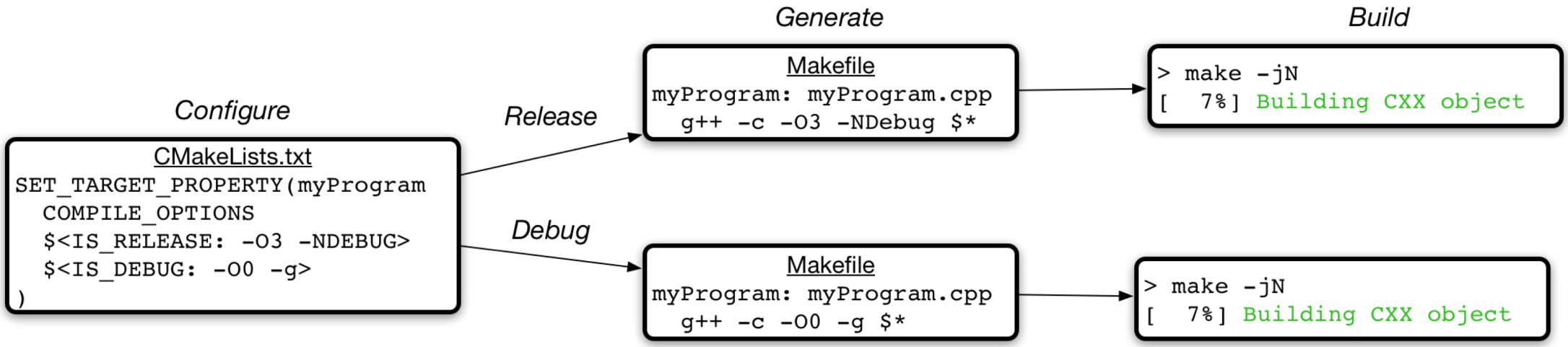
Spack dependency nightmare of Trilinos fixed soon?

Could be installed from a CMake build

Could be fetched/built as part of your project (Git submodule)

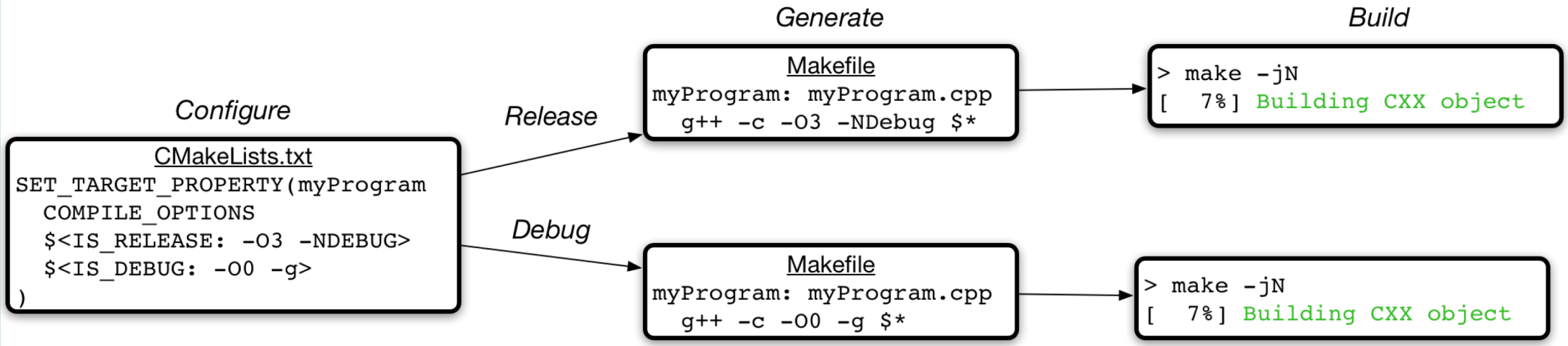
Separation between *configure*, *generate*, and *build* allows interaction between downstream and upstream

- ***Configure (Dynamic):*** Read options (e.g. `ENABLE_CUDA=ON`) and choose flags.
- ***Generate (Dynamic):*** Create Makefiles (or Ninja files, etc) based on configuration type (Debug/Release) and other variable values.
- ***Build (Static):*** Execute third-party tool to build dependency graph



Separation between *configure*, *generate*, and *build* allows interaction between downstream and upstream

- Generator expressions allow Kokkos to define build flags TO BE CONFIGURED LATER
  - `$<STREQUAL:$<TARGET_PROPERTY:PRECISION>, LOW>,--use_fast_math>`
  - Target property set by downstream library, flag set by Kokkos
- Iterative compilation, i.e. autotuning compiler pass order/selection, often finds better compiler optimization sequences for different classes of kernels
  - `$<STREQUAL:$<TARGET_PROPERTY:IC_CLASS>, FFT>, ${FFT_AUTOTUNED_FLAGS}>`



Spack handles some complexity, could handle more

- Spack compiler wrapper can add architecture or device flags to your project
- Spack could itself become a meta-build system that generates Makefiles/Ninja Files

```
class KokkosKernels(CMakePackage, CudaPackage):
        ...
        depends_on("kokkos")
```

```
class KokkosKernels(CMakePackage, CudaPackage):
        ...
        depends_on("kokkos", PUBLIC)
```

Modern CMake will enable best usage of modern C++ going forward to '20 and '23

- CMake could be de facto "standard" build system for modern C++ in future if you believe Reddit and CppCon
- Precompiled headers introduced into most recent CMake release
  - (Probably) no changes to your build system if Kokkos starts using precompiled headers.
  - Single call to `TARGET_LINK_LIBRARIES(Kokkos::kokkos)` handles all the complexity of the Kokkos interface
- C++ modules will (or won't) be coming soon
  - (Probably) no changes to your project build system if Kokkos starts using modules (code changes, though)
  - Single call to `TARGET_LINK_LIBRARIES(Kokkos::kokkos)` would handle all the complexity of the Kokkos interface and module dependencies

Harvard Business Review: Behavioral Economists #1 Reasons People Make Bad Decisions

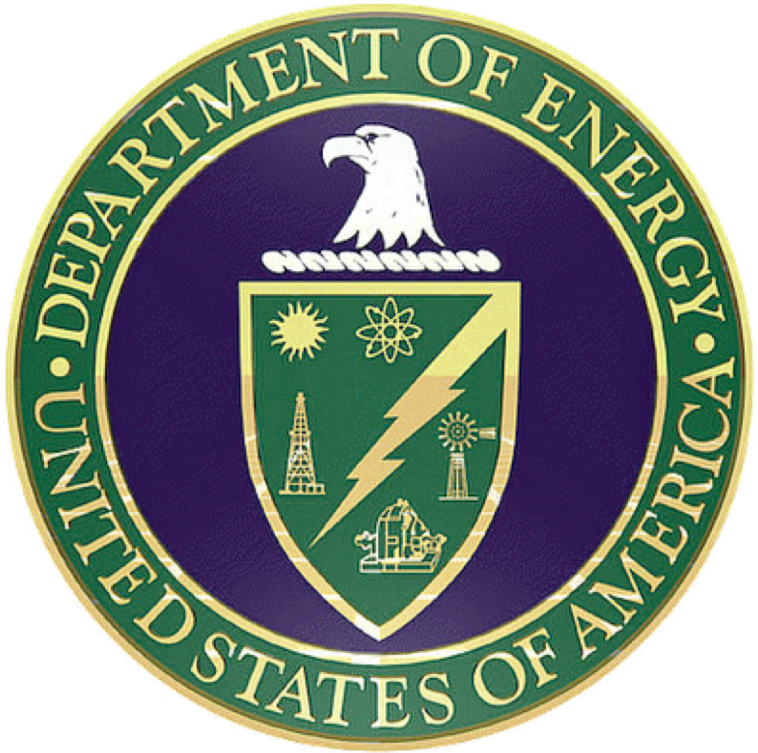
- ***Not anticipating unexpected events***
  - TARGET_LINK_LIBRARIES is smallest possible interface that allows Kokkos to be most agile without breaking behaviors users depend on (Hyrum's Law)
- ***Indecisiveness***
  - CMake has well-defined best practices (Professional CMake, Craig Scott)
- ***Remaining locked in the past***
  - Good luck with modules/precompiled headers using Automake
- ***Having no strategic alignment***
  - Does full stack modern C++ require program commitment to modern CMake?
- ***Isolation:***
  - Hard to bring tools from Kokkos ecosystem together if different build worlds
- ***Lack of technical depth***
  - CMake has a learning curve that is steeper than raw Makefiles
  - Make easy problems a bit more difficult but it makes hard problems tractable

Conclusions

- Kokkos is “generating” code for platforms from single C++ source, also needs to control your build system with single CMake source
- Build system design needs to define “building” and “using” requirements
  - Implement in CMake through targets and properties
- Targets and properties allow libraries to be modular components and a monorepo
- Targets and properties not only simplifies transitive dependencies, it allows *configurable* transitive dependences through generator expressions

Acknowledgments

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.



NNSA
National Nuclear Security Administration

Sandia National Laboratories